Numerical Computing w/Python, Lecture 2: NumPy and SciPy

Ramses van Zon

November 7, 2019

Python lists are not ideal for numerical arrays

For numerical work, the python-native lists aren't the ideal data type.

Lists can do funny things that you don't expect, if you're not careful.

- Lists are just a collection of items, of any type.
- If you do mathematical operations on a list, you won't get what you expect.
- These are not the ideal data type for scientific computing.
- Arrays are a much better choice, but are not a native Python data type.

```
>>> a = [1, 2, 3, 4]
>>> a
[1, 2, 3, 4]
>>>
>>> b = [3, 5, 5, 6]
>>> b
[3, 5, 5, 6]
>>>
>>> 2 * a
[1, 2, 3, 4, 1, 2, 3, 4]
>>>
>>> a + b
[1, 2, 3, 4, 3, 5, 5, 6]
>>>
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Lists vs. Arrays

Lists: optimized for flexibility

- Can hold any type
- Can grow
- Are one-dimensional
- Do not have out-of-the-box element-wise operations

Arrays: optimized for speed

- Single type
- Fixed size
- Multi-dimensional
- Have optimized element-wise operations

Arrays are what we want to use: Numpy

Almost everything that you want to do starts with NumPy.

- Contains arrays of various types and forms: zeros, ones, linspace, *etc.*
- linspace takes 2 or 3 arguments, the default number of entries is 50.

```
>>> import numpy
>>> numpy.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
>>> numpy.ones(5, dtype = int)
array([ 1,  1,  1,  1,  1])
```

```
>>> numpy.zeros([2,2])
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> numpy.arange(5)
array([ 0,  1,  2,  3,  4])
>>>
>>> numpy.linspace(1,5)
array([ 1.,  1.08163265,
        1.16326531,  1.24489796,
        .
        .
        4.67346939,  4.75510204,
        4.83673469,  4.91836735,  5. ])
>>> numpy.linspace(1, 5, 6)
array([ 1., 1.8, 2.6, 3.4, 4.2, 5.])
```

Specifying data types

```
>>> x = numpy.float32(7.4e-3)
>>> a = numpy.array([[1,2,3],[4,5,6]],dtype=numpy.float32)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]], dtype=float32)
>>>
>>> b = numpy.ndarray((2,3),dtype=numpy.float16)
>>> b
array([[ -1.51875000e+01,   5.11169434e-02,              nan],
       [  0.00000000e+00,  -3.12500000e+01,   4.35709953e-05]], dtype=float16)
```

Specifying data types

```
>>> x = numpy.float32(7.4e-3)
>>> a = numpy.array([[1,2,3],[4,5,6]],dtype=numpy.float32)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]], dtype=float32)
>>>
>>> b = numpy.ndarray((2,3),dtype=numpy.float16)
>>> b
array([[ -1.51875000e+01,   5.11169434e-02,              nan],
       [  0.00000000e+00,  -3.12500000e+01,   4.35709953e-05]], dtype=float16)
```

- Integers:
  `int8 int16 int32 int64 uint8 uint16 uint32 uint64`
  Number indicates number of bits.
- Floats of half, single and double precision: `float16 float32 float64`
- Complex numbers in single and double precision: `complex64 complex128`

Accessing array elements

Elements of arrays are accessed using square brackets.

- Like most languages, the first index is the row, the second is the column.
- Indexing starts at 0.
- You cannot assign values outside the index range (unlike e.g. in R).

Note: `import numpy as np` renames the `numpy` module to the shorter `np`.

```
>>> import numpy as np
>>> np.zeros([2, 3])
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
>>> a = np.zeros([2,3])
>>>
>>> a[1,2] = 1
>>> a[0,1] = 2
>>> a
array([[  0.,   2.,   0.],
       [  0.,   0.,   1.]])
>>>
>>> a[2,1] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 2 is out of bounds for axis
  0 with size 2
>>>
```

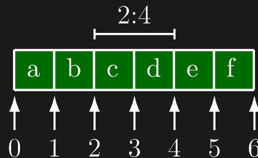
SciNet

Slicing arrays

You can select a subset of an numpy array by using an **index range** instead of a single number between square brackets. This is called **slicing**.

- An index range looks like “a:b”, e.g. “2:4”. So `a[2:4]` selects those elements of an array a.
- Read “2:4” as “from the beginning of the element at index 2, to the beginning of that at index 4”.
- Or read it as: index 2 is the first you get, index 4 is the first you do not get.
- Negative indexing is supported.
- If a third index is specified, it refers to the step size (“1:10:2”, for example).
- If no index is specifed, either “beginning” or “end” is assumed.

```
>>> a = np.array([1,2,3,4,5,6,7])
>>> print(a[2])
3
>>> print(a[2:4])
[3,4]
>>> print(a[::2])
[1,3,5,7]
```



Slicing arrays, continued

Elements in an array can also be selected using a boolean array. Boolean arrays can be created using a conditional expression.

```
>>>
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a > 2
array([False, False, False, True, True], dtype=bool)
>>> a[a > 2]
array([3, 4])
>>> a[(a % 2) == 0]
array([0, 2, 4])
>>>
```

The “%” symbol is the modulus operator.

1

Copying arrays

Copying array variables

```
>>> a = 10
>>> b = a
>>> a = 20
>>> a, b
(20, 10)
>>>
>>> a = np.array([[1,2,3],[2,3,4]])
>>> b = a
>>> a[1,0] = -10
>>> a
array([[1, 2, 3],
       [-10, 3, 4]])
>>>
>>> b
array([[1, 2, 3],
       [-10, 3, 4]])
```

Use caution when copying array variables. There's a 'sharing feature' here that is unexpected.

Copying array variables

```
>>> a = 10
>>> b = a
>>> a = 20
>>> a, b
(20, 10)
>>>
>>> a = np.array([[1,2,3],[2,3,4]])
>>> b = a
>>> a[1,0] = -10
>>> a
array([[1, 2, 3],
       [-10, 3, 4]])
>>>
>>> b
array([[1, 2, 3],
       [-10, 3, 4]])
```

Use caution when copying array variables. There's a 'sharing feature' here that is unexpected.
To turn off this 'sharing feature', use `copy()`:

```
>>>
>>> b = a.copy()
>>> a[1,0] = 16
>>> a
array([[1, 2, 3],
       [16, 3, 4]])
>>> b
array([[1, 2, 3],
       [-10, 3, 4]])
>>>
```

SciNet

2

Matrix arithmetic

Looping over arrays

- In Python, loops over arrays are performed over the first index.
- To go over all elements of a multidimensional array a without using nested loops, use `a.ravel()` or `a.flat` (or `a.flatten()` if you need a copy).

```
>>> a = np.array([1,2,3])
>>> for i in a:
...     print("element:", i)
element: 1
element: 2
element: 3
>>>
>>> a = np.array([[1,2],[3,4]])
>>> for i in a:
...     print("element:", i)
element: [1 2]
element: [3 4]
>>>
>>> for i in a.ravel():
...     print("element:", i)
element: 1
element: 2
element: 3
element: 4
>>>
```

SciNet

Shape and reshape

- NumPy allows you to modify the shape of an array once it already exists.
- Though, of course, you can only change the shape to one which contains the same number of elements.
- Also, note that `reshape` creates a new view of the array data, and doesn't change the shape of the original array.

```
>>> a = np.arange(8)
>>> a.shape
(8,)
>>>
>>> a.reshape([2,4])
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> a.reshape([2,4]).shape
(2, 4)
>>>
>>> a.reshape([2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of
  size 8 into shape (2,3)
>>>
```

Vector-vector & vector-scalar multiplication

1-D arrays are often called 'vectors'.

- When vectors are multiplied you get element-by-element multiplication.
- When vectors are multiplied by a scalar, you also get element-wise multiplication.

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>>
>>> b = np.arange(4.) + 3
>>> b
array([ 3.,  4.,  5.,  6.])
>>>
>>> c = 2
>>> c
2
>>>
>>> a * b
array([  0.,   4.,  10.,  18.])
>>> a * c
array([0, 2, 4, 6])
>>> b * c
array([  6.,   8.,  10.,  12.])
```

3

Matrix-vector multiplication

Peculiar matrix-vector multiplication

A 2-D array is sometimes called a 'matrix'.

- Matrix-scalar multiplication gives element-wise multiplication.
- Matrix-vector multiplication DOES NOT give the standard result!

```
>>> a = np.array([[1,2,3],[2,3,4]])
>>> b = np.array([1, 2, 3])
>>> a * b
array([[ 1,  4,  9],
       [ 2,  6, 12]])
```

Normal matrix-vector multiplication:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} \cdot b_1 + a_{12} \cdot b_2 + a_{13} \cdot b_3 \\ a_{21} \cdot b_1 + a_{22} \cdot b_2 + a_{23} \cdot b_3 \end{bmatrix}$$

Python matrix-vector multiplication:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} \cdot b_1 & a_{12} \cdot b_2 & a_{13} \cdot b_3 \\ a_{21} \cdot b_1 & a_{22} \cdot b_2 & a_{23} \cdot b_3 \end{bmatrix}$$

SciNet

Vector broadcasting

This peculiar multiplication is result of element-wise operations plus **broadcasting**.
Python will perform vector broadcasting if you perform a matrix-vector operation:

- Python will repeatedly apply the vector to the matrix.
- Python will not do this with vector-vector operations.
- The length of the vector must equal the last dimension of the matrix.
- By default it will do the application by row; use 'np.newaxis' to reshape the vector.

```
>>>
>>> a = np.zeros([2,3])
>>>
>>> a + np.arange(3)
array([[0, 1, 2],
       [0, 1, 2]])
>>>
>>> a + np.arange(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast
  together with shapes (2,3) (2,)
>>>
>>> a + np.arange(2)[:,np.newaxis]
array([[0, 0, 0],
       [1, 1, 1]])
>>>
```

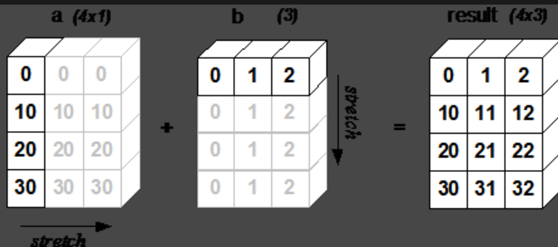
SciNet

Broadcasting rules

Dimensions are stretched if they are "compatible".

Start with the trailing dimensions, and move to the left.

If the dimensions are equal, or one of them is 1, or one is missing, they are compatible.



Matrix-matrix multiplication

Not surprisingly, matrix-matrix multiplication doesn't work as expected either, instead doing an element-wise multiplication like with vector-vector multiplication.

```
>>> a = np.array([[1,2,3],[2,3,4]])
>>> b = np.array([[1,2,3],[2,3,4]])
>>> a
array([[1, 2, 3],
       [2, 3, 4]])
>>> a * b
array([[ 1,  4,  9],
       [ 4,  9, 16]])
```

Normal matrix-matrix multiplication:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} & a_{11} \cdot b_{12} + a_{12} \cdot b_{22} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} \end{bmatrix}$$

Python matrix-vector multiplication:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \cdot b_{11} & a_{12} \cdot b_{12} \\ a_{21} \cdot b_{21} & a_{22} \cdot b_{22} \end{bmatrix}$$

How then to perform matrix algebra?

Since Python 3.5, the standard way is to use numpy arrays in conjuction with the @ product operator. Alternatively, you can use the dot function from SciPy.

```
>>> import scipy as sp
>>> a = np.array([[1,2,3],[2,3,4]])
>>> b = np.array([[1,2,3],[2,3,4]])
>>> a
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> a.T  # or a.transpose()
array([[1, 2],
       [2, 3],
       [3, 4]])
>>> a.T @ b  # or: sp.dot(a.T, b)
array([[ 5,  8, 11],
       [ 8, 13, 18],
       [11, 18, 25]])
>>> b @ a.T  # or: sp.dot(b, a.T)
array([[14, 20],
       [20, 29]])
>>> c = np.arange(3) + 1
>>> a @ c # or: sp.dot(a,c)
array([14, 20])
>>>
```

SciNet

4

Linalg Submodule of SciPy

The linalg submodule

The linalg submodule of SciPy contains useful functions for matrix algebra.

- Typical matrix functions: inv, det, norm...
- More advanced functions: eig, SVD, cholesky...
- Both NumPy and SciPy have a linalg module. Use SciPy, because it is compiled with optimized BLAS/LAPACK support.

```
>>> import numpy as np
>>> import scipy as sp
>>> import scipy.linalg as linalg
>>> a = np.array([[1,2,3], [3,4,5], [1,1,2]])
>>> linalg.det(a)
-2.0
>>> sp.dot(a, linalg.inv(a))
array([[1.00000000e+00, 1.11022302e-16, 0.00000000e+00],
       [4.99600361e-16, 1.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 1.00000000e+00]])
>>>
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Solving systems of equations

The linalg submodule of scipy comes with an important function: solve.

linalg.solve is used to solve the system of equations $Ax = b$.

```
>>> a = np.array([[1,2,3], [3,4,5], [1,1,2]])
>>> a
array([[ 1,  2, 3],
       [ 3, 4, 5],
       [ 1, 1, 2]])
>>> b = np.array([3, 4, 2])
>>> b
array([3, 4, 2])
>>>
>>> x = linalg.solve(a, b)
>>> x
array([-0.5, -0.5, 1.5])
>>>
```

Here

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} x[0] \\ x[1] \\ x[2] \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

is solved by

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} -0.5 \\ -0.5 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

5

Statistics

Statistics

SciPy contains all of the statistical functions that you'll probably ever need.

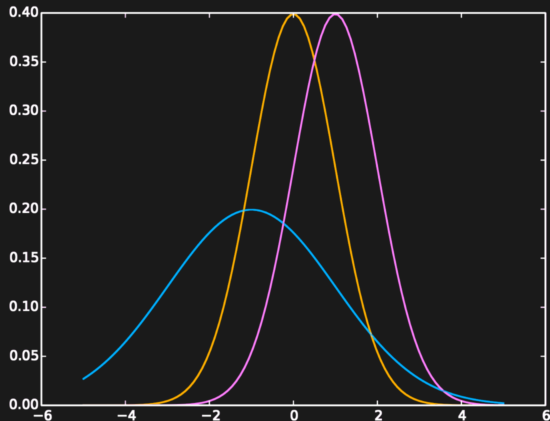
- The scipy.stats module is based around the idea of a 'random variable' type.
- A whole variety of standard distributions are available:
  - Continuous distributions: Normal, Maxwell, Cauchy, Chi-squared, Gumbel Left-scewed, Gilbrat, Nakagami, . . .
  - Discrete distributions: Poisson, Binomial, Geometric, Bernoulli, . . .
- The random variables have all of the statistical properties of the distributions built into them already: cdf, pdf, mean, variance, moments, . . .

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Normal statistics

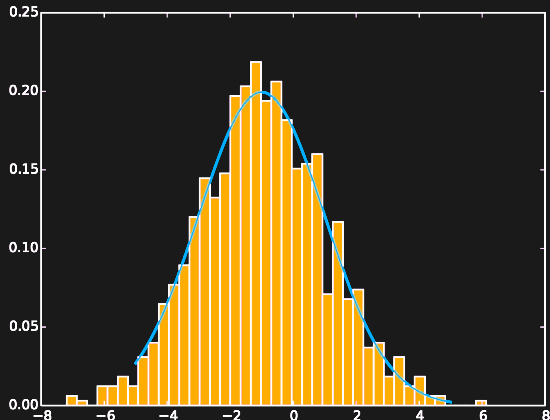
```
>>> import numpy as np, scipy as sp
>>> import matplotlib.pyplot as plt
>>> import scipy.stats as stats
>>> x = np.linspace(-5, 5, 100)
>>> plt.plot(x,stats.norm.pdf(x))
>>> plt.plot(x,stats.norm.pdf(x,loc=1))
>>> plt.plot(x,stats.norm.pdf(x,loc=-1,scale=2))
>>>
```

All continuous distributions take `loc` and `scale` as keyword parameters to adjust the location and scale of the distribution. In general the distribution of a random variable `X` is obtained from `(X - loc) / scale`. The default values are `loc = 0` and `scale = 1`.



Normal statistics, continued

```
>>>
>>> stats.norm.mean(loc = -1, scale = 2)
1.0
>>> stats.norm.std(loc = -1, scale = 2)
2.0
>>> stats.norm.moment(3, loc = -1, scale = 2)
-13.0
>>> samples = stats.norm.rvs(size = 1000,
...                          loc = -1, scale = 2)
>>> plt.hist(samples, bins=41, density=True)
>>>
>>> plt.plot(x,
...     stats.norm.pdf(x, loc = -1, scale = 2),
...     'c', linewidth = 2)
>>>
```



Setting the seed

Sometimes you need consistency in your randomness:

- Pseudo-random numbers are generated from an initial 'seed'.
- This seed generates the first number, which is then used as the seed for the second number.
- If you need consistency in your random numbers (for debugging, for example), you can set the seed explicitly so that you get the same random numbers every time.
- Be careful using this for production!

```
>>> stats.norm.rvs()
1.74481176421648
>>> stats.norm.rvs()
-0.7612069008951028
>>>
>>> np.random.seed(1)
>>> stats.norm.rvs()
1.6243453636632417
>>>
>>> sp.random.seed(1)
>>> stats.norm.rvs()
1.6243453636632417
>>>
>>> import random as rd
>>> rd.seed(1)
>>> stats.norm.rvs() # stats unaffected
-0.6117564136500754
```

Random versus numpy.random

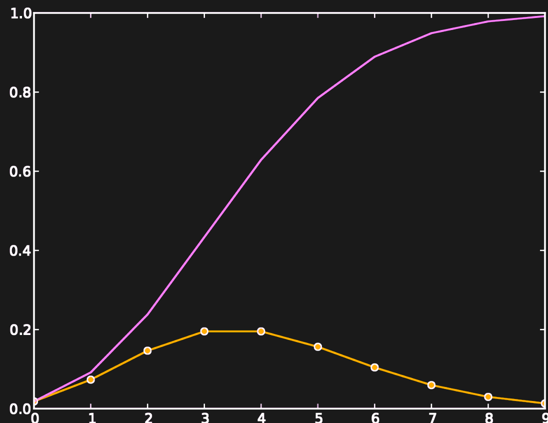
You may notice that there are several random packages: `random`, `numpy.random` and `scipy.random`.
What's the difference?

- scipy.random and numpy.random are the same.
- The `random` package is not connected to the others.
- The numpy.random package affects numpy and scipy routines; the `random` package does not.
- All use the same algorithm (Mersenne Twister).
- The `random.seed()` is thread safe, while numpy's and scipy's `random.seed()` are not.
- The `numpy.random` package contains more functionality.
- Unless you need your code to be thread-safe (rarely in python), use `numpy.random`.

SciNet

Statistics, a discrete example: Poisson

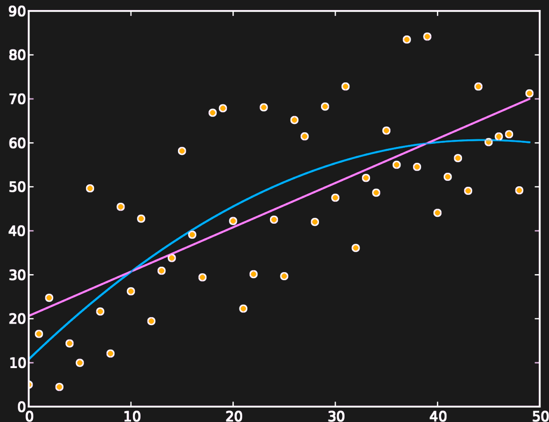
```
>>>
>>> x = np.arange(10)
>>> plt.plot(x, stats.poisson.pmf(x, 4),'o-')
>>> plt.plot(x, stats.poisson.cdf(x, 4))
>>> stats.poisson.mean(4)
4.0
>>> stats.poisson.var(4)
4.0
>>>
```



Note that discrete distributions have Probability Mass Functions (PMF) instead of Probability Density Functions (PDF).

Polynomial fitting

```
>>>
>>> x = np.arange(50.)
>>>
>>> y = x + 50.0 * np.random.random(50)
>>>
>>> plt.plot(x, y, 'o')
>>>
>>> fit = np.polyfit(x, y, 1)
>>> fit
array([  1.0073584,  20.64695036])
>>> plt.plot(x, np.polyval(fit, x))
>>>
```




```
>>> fit = np.polyfit(x, y, 2)
>>> fit
array([ -0.02520835, 2.24256777,  10.76527546])
>>> plt.plot(x, np.polyval(fit, x))
```

Further numerical functionality in SciPy

There is a lot functionality more in SciPy and its subpackages, e.g.:

- optimization
- (even more) linear algebra
- integration
- interpolation
- special functions
- fast fourier transforms
- signal and image processing
- solvers for ordinary differential equations.
- ...

SciNet

6

Assignment

Assignment 1

1. Write a script to find a rough approximation to the **location of the minimum** of the function

$$y(x) = x - \frac{15x}{1+x}$$

   by computing the values of the function for $N = 10$ evenly spaced values of x between 0 and 10, and determining the x value corresponding to the minimum y.

   Next, the script should successively increase the number of evenly spaced x-values between 0 and 10, taking $N = 10^2$, 10^3, 10^4, 10^5 and 10^6.

   The script should print the x value of the minimum and its accuracy for each N.

2. Find a better way to do this using SciPy. Check the accuracy.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO